Abstract of the Thesis

Genetic Programming for Image Compression

by

Thomas M. Wolfley

Master of Science in Computer Science

University of California, Los Angeles, 2010

Professor Demetri Terzopoulos, Chair

Genetic Programming is used to evolve expressions that compactly represent target images. More specifically, we develop a method for creating images by evaluating program expressions consisting of various functions and operations. Applying genetic programming to these program expressions, which serve as "image genomes," our genetic algorithm can evolve image representations that closely match given target input images. The evolved image representations form compressed, lossy representations of the original target images.

Table of Contents

1. Introduction 1
1.1 Thesis Overview 5
2. Related Work 6
3. Methodology 8
3.1. Genotype 8
2.2. Phenotype 9
3.3. Fitness Function 12
3.4. Algorithm 12
3.5. Mutation and Crossover 13
3.6. Subdivision 16
3.7. System parameters 17
4. Experiments and Results 20
4.1. No Subdivision 20
4.2. Uniform Subdivision 21
4.3. Variable Subdivision 21
5. Conclusion and Future Work 27
5.1. Conclusion 27
5.2. Future Work 27
References 30

List of Figures

Figure 1. User-created images and the expressions that produce the images 4
Figure 2. hypot(sin(x*y),x) - 1 8
Figure 3. 8
Figure 4. Generating a Phenotype 9
Figure 5. hypot(sin(x*y),x)-1, range (-10, 10) 10
Figure 6. sin(x*y) - y / x, range (-10, 10) 10
Figure 7: 3-dimensional graph of sin(x+y), range (0, 1) 11
Figure 8. 2-dimensional height map of sin(x+y), range (0, 1) 11
Figure 9.Original genotype: x*x + y*y 14
Figure 10. x*x + y*y 14
Figure 11. 14
Figure 12. 14
Figure 13: x*x + y*y - 10 14
Figure 14. x*x + y 14
Figure 15. cos(x)+cos(y) 15
Figure 16. cos(x*y) 15
Figure 17. cos(x)+y 15
Figure 18. cos(x*cos(y)) 15
Figure 19. Variable Subdivision algorithm 16
Figure 20. hypot(x+y, y) 19
Figure 21. Target Image 20
Figure 22. No subdivision 20
Figure 23. Uniform subdivision 20
Figure 24. Improvement over time for no subdivision 21
Figure 25. Accepted fitness, size, running time, and RMS error for each image 25
Figure 26. Results from JPEG 26
Figure 27. Cat; First run, 20,773 bytes 26
Figure 28. Cat; second run, 21,247 bytes 26

Acknowledgements

This thesis would not have been possible without the excellent guidance of my committee chair, Professor Demetri Terzopoulos, whose work, teaching, and advice provided me with much of the inspiration necessary to write it. I am greatly thankful for his support and efforts.

I would also like to thank my committee members, Professors Michael Dyer and Petros Faloutsos, who have also supported me throughout this work.

1. Introduction

Genetic Algorithms [Holland, 1988] are methods for finding solutions to optimization and search problems by using techniques inspired by natural evolution. Typically, genetic algorithms have several components that are unique to each algorithm:

1. **Problem:** A problem to be solved with a very large solution space.
2. **Genotype:** A means by which a solution can be internally represented—e.g. a series of numbers, a list of bits, or a tree structure.
3. **Phenotype:** The result of processing the genotype. This can often be expressed in different ways. It is usually useful to consider the phenotype as a human-understandable view of the solution that a particular genotype specifies.
4. **Innovation:** A means to modify the genotype by mutation and crossover.
5. **Fitness:** A fitness function to evaluate the quality of the genotype as a solution to the given problem.

Running a genetic algorithm involves a series of steps:

1. Initialize a population of N individuals
2. Evaluate the fitness of each member of the population
3. Select N members of the population based on fitness, using a selection function; some will be selected more than once
4. Apply reproduction, mutation, and crossover on the selected members to create a new population of N members
5. Repeat steps 2 through 4 until some termination criterion is reached—normally either a satisfactory solution or a maximum number of generations.
6. Choose the individual with the highest fitness over all generations as the result of the computation.

Genetic programming [Koza, 1992] is a particular category of genetic algorithms in which the genotype is a computer program represented internally as an Abstract Syntax Tree (AST). It is a general technique to computationally evolve programs that exhibit the desired properties expressed by the fitness function. The algorithm may employ any existing language, or it can use a specific language created for the purpose of the algorithm. Genetric programming has been used to solve many problems in various fields, including systems modeling, machine control, optimization, and scheduling, design, and signal processing [Willis et al., 1997].

Our contribution in this thesis is to apply genetic programming to image compression. To our knowledge, genetic programming has not yet been employed successfully for image, audio, or video compression.

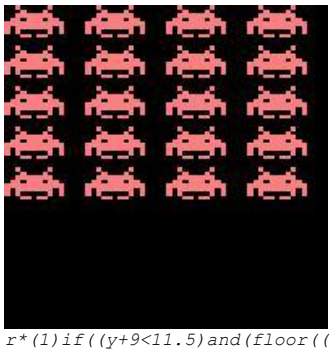
Our first development, which we introduce here as motivation for our work, is a method to synthesize images using program expressions. This is essentially the genotype for our genetic programming algorithm. We implemented it separately on a website[1] in order to determine its power as a method of image generation. An anonymous user can visit the website, type in an expression (the genotype), and see the image (the phenotype) that results from processing that expression. The image is saved to the server and the gallery of all created images can be browsed. Users have so far created over 30,000 images, many of which are rather interesting. Figure 1 shows examples of the images that users have created.

These examples clearly demonstrate that our approach is capable of generating many kinds of 2D graphics by combining a variety of basic functions and operations. Furthermore, consider the size of the expressions that generate the example images. All that need be stored to save these images to a file is their associated expressions, which are shown underneath each image in Figure 1. These expressions can be further simplified and com-

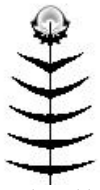
[1] http://fragsworth.com/image_functions

pressed, so that each of these images can potentially be represented in a very small space—just a few dozen bytes. The method we used to generate images in our genetic programming system employs a simplified version of these sorts of expressions, as is explained in Section 3.

In light of these motivational observations, our objective in this thesis is to develop a practical method for evolving lossy representations of target images through the use of genetic programming. A target image is selected, and the genetic programming algorithm attempts to find a set of programs (similar to those in the captions of Figure 1), which render the target image when evaluated over an array of pixels.



```
r*(1)if((y+9<11.5)and(floor((
x)*3)%15,floor((y+10)*4)%10)i
n((2,0),(8,0),(3,1),(7,1),(2,
2),(3,2),(4,2),(5,2),(6,2),(7
,2),(8,2),(1,3),(2,3),(4,3),(
5,3),(6,3),(8,3),(9,3),(0,4),
(1,4),(2,4),(3,4),(4,4),(5,4)
,(6,4),(7,4),(8,4),(9,4),(10,
4),(0,5),(2,5),(3,5),(4,5),(5
,5),(6,5),(7,5),(8,5),(10,5),
(0,6),(2,6),(8,6),(10,6),(3,7
),(4,7),(6,7),(7,7)))else(-1)
```

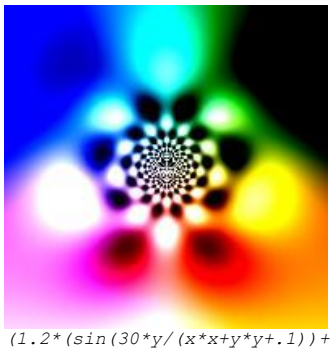


```
pow(sin(sqrt(pow((atan2(x,y)
*y*y/10),2)+pow(sqrt(x*x+y*y
)*2,2))*2+abs(sin(atan2(sqrt
(x*x+y*y),(atan2(x,y)*1))*10
))),7)*9/(pow((atan2(x,y)),2
))+x*x*2+y*y
```

```
((r+g)*(1-
2*((1>pow((atan2(y,x)*10/pi)%
2.-
1,2)+pow((100/(x*x+y*y+1))%2-
1,2)>0.8)+(hypot((atan2(y,x)*
10/pi)%2-
1.3,(100/(x*x+y*y+1))%2-
.8)<.2)+(hypot((atan2(y,x)*10
/pi)%2-
.7,(100/(x*x+y*y+1))%2-
.8)<.2)+(.6>hypot((atan2(y,x)
*10/pi)%2-
```

```
(lambda(a),e,d,x,y,k,l,w,abs,c,s,
atan2,pi,pow:d(d(e(a-
2.5,l+.7,.75,r-g*.45-
b)+(a<2.5)*(y>-4)*(1-
e(k,l,2.2,1))*e(k,l,2.5),d(d(e(k,
l,3.3,1),e(k,l,3.5)),d(d(e(x,w,3.
2,1),e(x,w,3.4)),d(e(x,y-
3,4.0,1),e(x,y-3,4.2))+d(e(a-
4,y+5,1,1),e(a-
4,y+5,1.2))+d(d(e(x/3,y-
9,1.8,1),e(x/3,y-9,2)),d(d(e(x-
3,y+8,.7,1),e(x-3,y+8,.9)),d(-
```

```
(hypot(x,y+2)<4)*((2*(ifte(si
n(asin(((x*.24/sqrt(1-
(asin(((((y+2)*.25)**2)+11)%2
-1)*2/pi)))+11)%2-
1)*3*pi)*(sin(asin((((y+2)*.2
5)+11)%2-1)*3*pi)),r-g-
b,1)))-
(1.8*(hypot(x+2,y+6)/5)))+(y>
3.5)*(1-1.2*(1-((x-
2)**2*.031+((y-8)**2))))
```



```
(1.2*(sin(30*y/(x*x+y*y+.1))+
cos(40*x/(x*x+y*y+.1)))+0.4*y
*r-0.4*x*b)
```

Figure 1. User-created images and the expressions that produce the images

1.1 Thesis Overview

The remainder of this thesis is organized as follows: Section 2 presents a review of relevant work. In Section 3, we will describe the specific details of how we used the genetic programming approach to compress images. This will include a detailed discussion of the genotype, phenotype, and fitness function. Additionally, we will describe the general procedure of the genetic programming algorithm; including the methods of mutation, crossover, and reproduction, with some examples applied to our genotypes and phenotypes. We will also describe some additional techniques to make the algorithm produce agreeable results. In Section 4, we will present our experiments and discuss the results of our initial attempts and progressive improvements, and show some of the best compressed images which were produced after several runs of the algorithm. We conclude in Section 5 with a discussion of the future work to be done with regard to this technique.

2. Related Work

JPEG [Wallace, 1991] is the dominant standard for image compression today. It can typically produce lossy compression ratios of 10:1 or better with acceptable image quality. Through the use of the discrete cosine transform [Ahmed et. al, 1974], JPEG takes advantage of several facts about human vision. Since the eye is less sensitive to color detail (as opposed to brightness detail), color detail can be suppressed to a larger extent than brightness without substantial loss of perceptual quality. Human vision is also more sensitive to small brightness changes over a large area, rather than to the strength of brightness changes over a small area, and this can be taken advantage of as well—by reducing the accuracy of brightness over high frequency areas. JPEG is fast during both compression and decompression.

Fractal Image Compression [Barnsley et. al, 1993] is a method to produce fractals that represent a target image after one or more iterations of the fractal. Fractal image compression takes advantage of self-similarities within images and works best on images that have many structured patterns. Although it does not use a genetic algorithm, fractal image compression can take significant time during compression and is fast during decompression.

Sims [1991] used an artificial evolution method very similar to our own in order to evolve artistic images, textures, structures, and motions for use in computer graphics. Lisp expressions were used to represent the evolved graphical elements. Subjective interactive selection was used to assess the fitness of the individuals in the system, which allowed users to "breed" progressively more interesting results. Our method of user-created images presented in Section 1 and Figure 1 is inherently analogous to this, but it was non-algorithmic. The users themselves naturally tend to choose images that they like from the existing population, and apply their own—oftentimes intelligent—mutations to

those images.

Alsing [2008] used a genetic algorithm to evolve an image of the Mona Lisa. The genotype in this experiment was a string that represented a series of semi-transparent polygons. The corner location, color, and transparency of each polygon can vary over the course of the algorithm. The result was a recognizable image of the Mona Lisa, but there was no discussion of the size of its representation or the potential for using this method as a form of image compression.

3. Methodology

This section will cover the details of our implementation of the genetic programming algorithm. We will discuss the fundamental details of our algorithm—genotype, phenotype, fitness function, mutation and crossover—as well as our system's unique parameters and techniques.

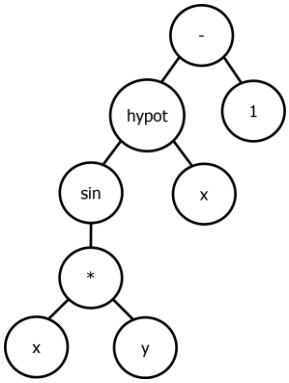


Figure 2.
*hypot(sin(x*y),x) - 1*

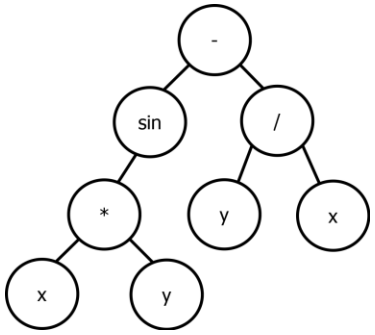


Figure 3.
*sin(x*y) - y / x*

3.1. Genotype

Python [Rossum, 2008] was our programming language of choice for our implementation due to its ubiquity and simplicity. A major advantage of this language is the existence of a well-tested, built-in parser and compiler, which we used heavily.

The individuals in the system have genotypes that are Python expressions. These expressions can contain anything from a selected list of allowed Python programming constructs, as well as two additional variables—`*x*` and `*y*`. The allowed programming constructs include a list of functions, and any arithmetic operator (+, -, /, *). For example,

individuals can have the following genotypes:

- `hypot(sin(x*y),x) - 1`
- `sin(x*y) - y / x`

The above expressions are represented as simple text, but they can also be represented as *Abstract Syntax Trees* (ASTs) (Figures 2 and 3). By converting them to ASTs, we are able to treat the expressions as tree structures, which is necessary for applying mutation and crossover to the genotypes.

Python is by no means the only option in this regard and, in fact, due to the limitations that we impose on the allowed language constructs, many languages which support the evaluation of mathematical expressions can be used equally well, including Lisp.

2.2. Phenotype

The phenotype of an individual in the system is determined by evaluating their genotype over each element of a 2-dimensional array to produce an image. A high level description of generating a phenotype is as follows:

```
f = genotype as a function of x and y
for i in the width of the image:
    for j in the height of the image:
        x = scale(i, 0, width, 0, 1)
        y = scale(j, 0, height, 0, 1)
        z = f(x, y)
        brightness = floor( scale(z, -1, 1, 0, 255) )
        store brightness in pixel position (i, j)
```

Figure 4. Generating a Phenotype

We scale i and j such that *x* and *y* ranges between either (0, 1) or (-10, 10) in the images

that follow. The *z* range is scaled between -1 and 1 for the brightness of the image at a given pixel. For instance, if $f(x, y) \leq -1$, the brightness of the image at (i, j) is 0. If $f(x, y) \geq 1$, the brightness of the image at (i, j) is maximized (i.e., 255 for 8-bit gray-scale images). These ranges were chosen after subjectively viewing a variety of expressions at different ranges.

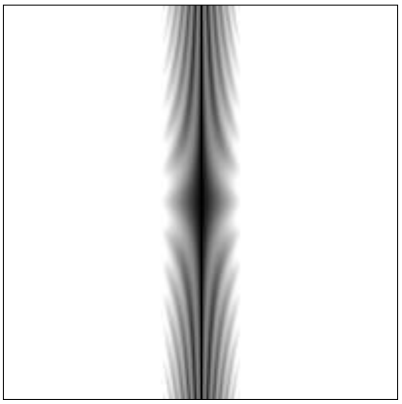


*Figure 5. hypot(sin(x*y),x)-1, range (-10, 10)*

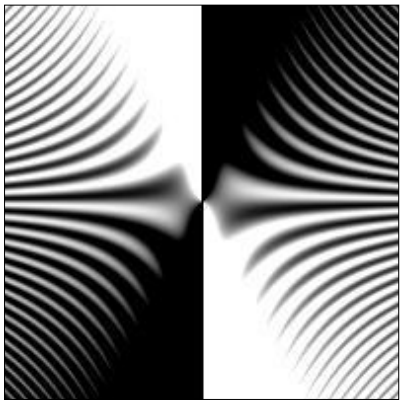


*Figure 6. sin(x*y) - y / x, range (-10, 10)*

The results of this calculation for the genotypes chosen earlier are black and white images (Figures 5 and 6). Color can be produced using the same process three times, once for each of red, green, and blue channels, and merging the results. In our system, however, we evolved only greyscale images.

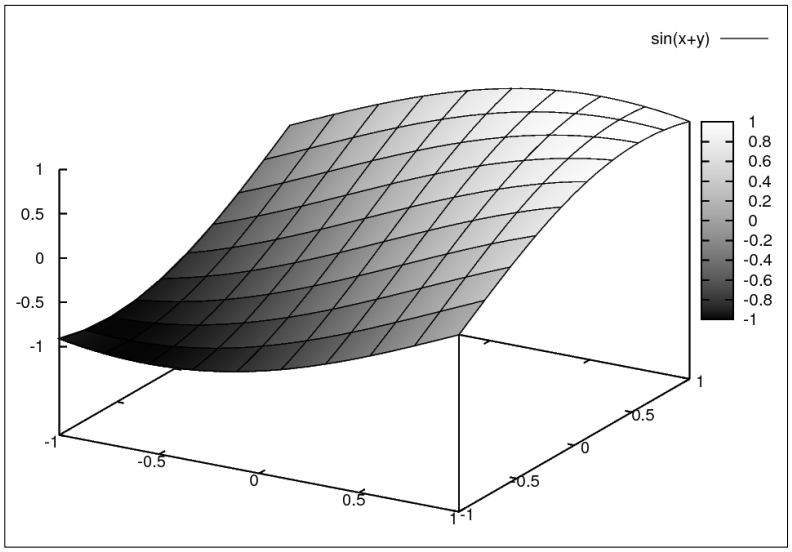


Figure 7: 3-dimensional graph of sin(x+y), range (0, 1)

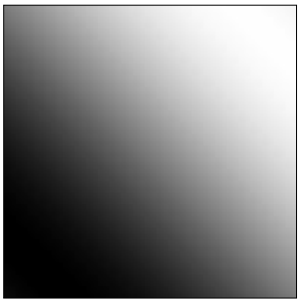


Figure 8. 2-dimensional height map of sin(x+y), range (0, 1)

The resulting images may often appear to be visually similar to iterative function systems, or fractals [Barnsley et. al, 1993], but it should be stressed that these are not fractals. They are not products of an iterative approach. They are conceptually much simpler, and can best be thought of as height-maps of a function over two variables (Figures 7 and 8).

Like our choice of language, our choice of image representation was by no means the only option available to us. This image representation was chosen based on subjectively ob-

serving the results of many human-generated expressions. However, any process that produces images from a tree structure similar to ASTs can be used equally well, and approaches that use fractals or other methods are certainly possible.

3.3. Fitness Function

For a selected target image, the fitness function of an evolved image in the simulation is the pixel-wise root-mean-square difference between the evolved image and the target image:

$$fitness = \sqrt{\frac{\sum_{i=0}^{w}\sum_{j=0}^{h}(orig_{i,j} - result_{i,j})^2}{w * h}}$$

This fitness function was chosen for simplicity and efficient computation. The fitness function could potentially include many other parameters. Of particular interest would be those that take into account the sensitivity and limitations of human vision, among them to brightness variation over an area.

3.4. Algorithm

The following is a high-level description of our genetic programming algorithm:

1. Create the initial population of N randomly generated individuals.
2. Calculate the fitness of each individual—i.e., render the individual's function as an image, and determine the root-mean-square difference between the rendered image and the target image.
3. Create a new generation through reproduction—given a probability distribution function, choose N individuals from the current generation; perform mutation and crossover on each individual.
4. Repeat steps 2 and 3 until the termination criteria are met.
5. The individual with the highest fitness over all the generations is chosen as the result.

Genetic algorithms (and genetic programming) can often have very long running times.

Some algorithms take days or even weeks to finish. The evolved result, however, which in our case is an expression, can be computed virtually instantaneously.

3.5. Mutation and Crossover

Mutation and crossover are the possible methods that can be applied to innovate new individuals in the genetic programming system. When a new generation is created, a member from the old population is selected and one of these two options (mutation or crossover) is chosen at random with equal probability.

Mutation amounts to a choice of one of these five options:

- **Add a node:** A node is randomly generated, selected from the pool of possible programming nodes. The new node is inserted into the AST at a random position between two existing parent and child nodes, and a leaf node is randomly generated to attach to it.
- **Delete a node:** A non-leaf node is chosen at random to delete. Its parent and child nodes are merged. If it has more than one child, one is randomly selected to merge with the parent and the remaining nodes are discarded.
- **Change a node:** A non-leaf node is randomly selected to change into another node that can handle the same number of children.
- **Modify constants:** Each constant in the tree (leaf nodes that are numeric) has a probability of being slightly modified by adding to or subtracting from its value within a certain range. This is a non-standard approach to mutation, which we introduced based on several assumptions. Constants are different from other nodes in that they may be modified slightly in a non-discrete manner—changes to constants should less drastically affect the image than changes to nodes. Additionally, increasing or decreasing the values of constants should often improve the resulting image.
- **No change:** The selected individual is copied over to the next generation without modification.

Figure 9 shows a base genotype and Figure 10 shows its phenotype. Examples of the effects of adding a node (Figures 11 and 12) and deleting a node (Figures 13 and 14) are shown as well. These figures serve to illustrate the transformations that can occur to an image and its representation.

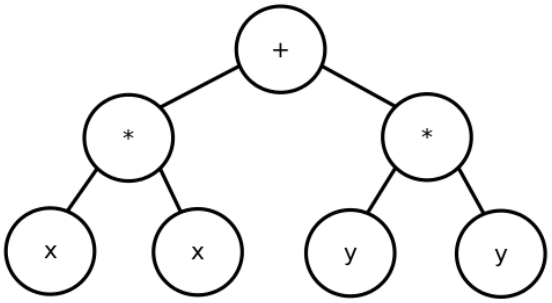


*Figure 9.Original genotype: x*x + y*y*

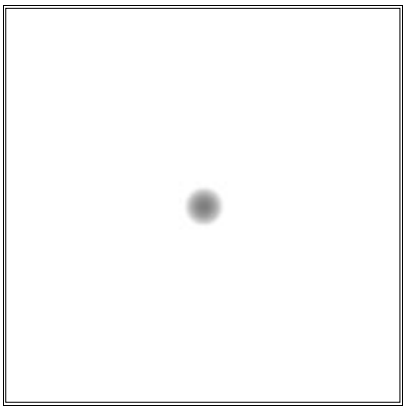


*Figure 10. x*x + y*y*

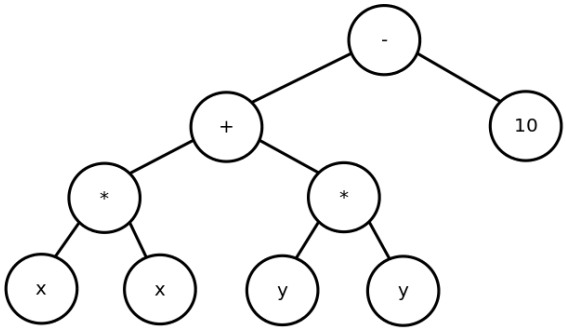


Figure 12.
*Adding a Node: x*x + y*y - 10*

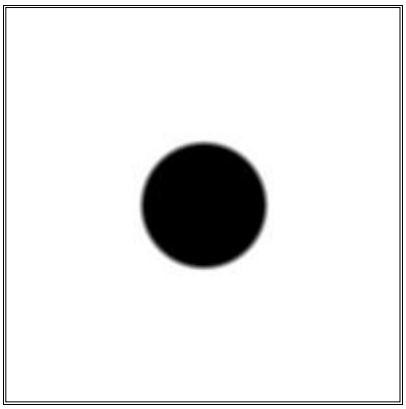


*Figure 13: x*x + y*y - 10*

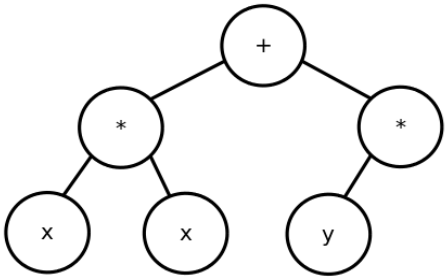


Figure 11.
*Deleting a node: x*x + y~~*y~~*

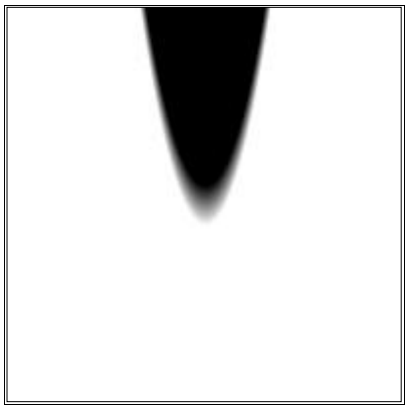


*Figure 14. x*x + y*

Crossover is performed on an individual by selecting an additional member of the old population, and swapping two sub-trees between them. Both individuals are inserted into the new population. Crossover on two individuals is shown in the following figures.

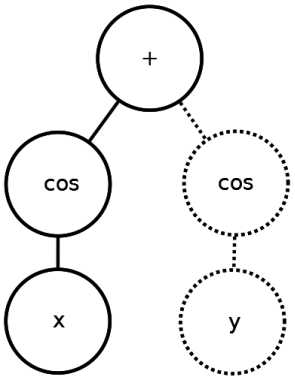


Figure 15. cos(x)+cos(y)

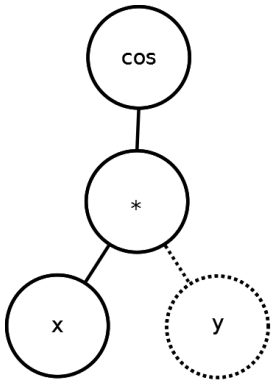


*Figure 16. cos(x*y)*

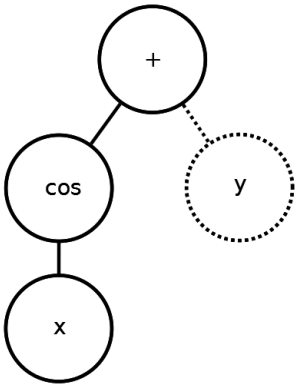


Figure 17. cos(x)+y

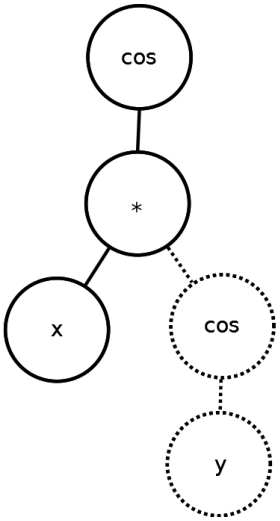


*Figure 18. cos(x*cos(y))*

The two individuals (Figures 15 and 16) have their sub-trees "cos(*y*)" and "*y*" swapped, respectively. The results are Figures 17 and 18.

3.6. Subdivision

In Section 4, we will explain that a single function cannot easily represent an entire image. Because of this, we chose to divide the image into sub-images and run the genetic algorithm on each sub-image. We refer to this as *uniform subdivision*. By dividing the image into several smaller sub-images, the algorithm can attempt to match these smaller, simpler sub-images rather than the entire image.

A more adaptive approach was chosen, however, for the final result. We refer to this as *variable subdivision.* The genetic programming algorithm is initially run on an image in its entirety. If the image cannot be matched adequately, it is subdivided into 4 smaller sub-images and the process is repeated recursively. By varying the adequacy threshold, we can, to some extent, determine how accurately the generated expressions will represent the target image.

Variable subdivision is performed recursively (Figure 19). This results in a tree structure that can be traversed to generate the final image.

```
function tree(bounding_box, full_image):
    node = TreeNode # a TreeNode is either a result or a set of children
    result = run_bounded_gp(bounding_box, full_image)

    if result.fitness > acceptable_fitness and bounding_box.size >
min_box_size:
        bboxes = split_into_quadrants(bounding_box)
        for bbox in bboxes:
            node.children.append( tree(bbox, full_image) )
    else:
        node.result = result
    return node
```

Figure 19. Variable Subdivision algorithm

The "`acceptable_fitness`" variable allows us to control the quality of the resulting image. If `acceptable_fitness` is increased, the image quality will be reduced, and vice versa. The image will be divided into quadrants if it does not meet the acceptable fitness value. A minimum box size can be specified to prevent the algorithm from recursing too deeply, regardless of the current fitness. The most extreme case would be to allow the algorithm to match functions to individual pixels of the image. If this happens, it could be desirable to combine other techniques into the system that reduce the bloat of having large expressions represent small portions of the image.

3.7. System parameters

This section discusses the parameter choices that we made in our system.

The population size is 80, and the system will attempt to produce an acceptable fitness in up to 500 generations, at which point the image will subdivide or terminate recursion due to the current subdivision size being too small.

Trees are limited to 40 nodes, and the depth of trees is limited to 20. Individuals that exceed these limits will only delete nodes during mutation. The limitations can be exceeded temporarily, when crossover or mutation through node addition occurs.

The selection function, which chooses an individual from the previous generation, is based on the exponential distribution:

$$f(x, \lambda) = \frac{-ln(x) * P}{\lambda}$$

where *P* is the population size (80). The specific value for λ is 15 and *x* is randomly chosen uniformly between 0 and 1. The result of this function is a number whose floor value can be considered to be a choice within the population. There is higher probability of se-

lecting a member that is ordered lower in index, which is appropriate for how the individuals are sorted based on fitness.

When constants are generated, they are chosen randomly, as floating-point numbers uniformly distributed in the range (-10, 10). When the constant modification operation is performed, constants are changed by adding a random value uniformly distributed in the range (-1, 1).

The allowed operators were selected as follows:

Double-child operators:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- hypot()

Single-child operators:

- sin()
- cos()
- tan()
- bound()

Leaf nodes:

- Constants
- Variable "*x*"
- Variable "*y*"

When nodes are added to a tree, the choice of node is selected at random from the set of Single-child and Double-child operators with uniform probability. This list is much shorter than the full capability of the language, but it was a choice we made to reduce the complexity of the system.

The hypot(*a*,*b*) operation can be described as the hypotenuse of a right triangle, or the Euclidian distance between the origin [0,0] and [*a*,*b*]. This operation was added for its tendency to create ellipsoidal gradients (Figure 20), and the assumption that these would generally be helpful mutations to an image.

The bound() operation is a non-standard function that we created. It imposes bounds of -1, 1 to its argument. This operation was added to enable mutations to have reasonable effects on expressions whose results go far beyond the range of image brightness (the *z*-axis).

Figure 20. hypot(x+y, y)

4. Experiments and Results

We have run several experiments with our system. In the following three subsections, we will report on the results that we have obtained with no subdivision, uniform subdivision, and variable subdivision, respectively.

4.1. No Subdivision

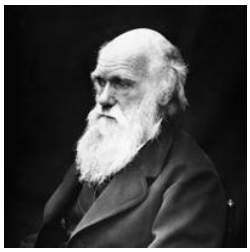


Figure 21. Target Image

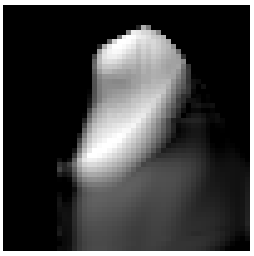


Figure 22. No subdivision

Figure 23. Uniform subdivision

Initially, we attempted to evolve a single function to match an entire image. This proved to produce visibly poor results, as shown in Figure 22.

It may be possible to achieve satisfactory results with a single function, but our attempts with various parameter changes did not yield anything better. The system gets caught in local minima without any significant improvement over many generations (Figure 24).

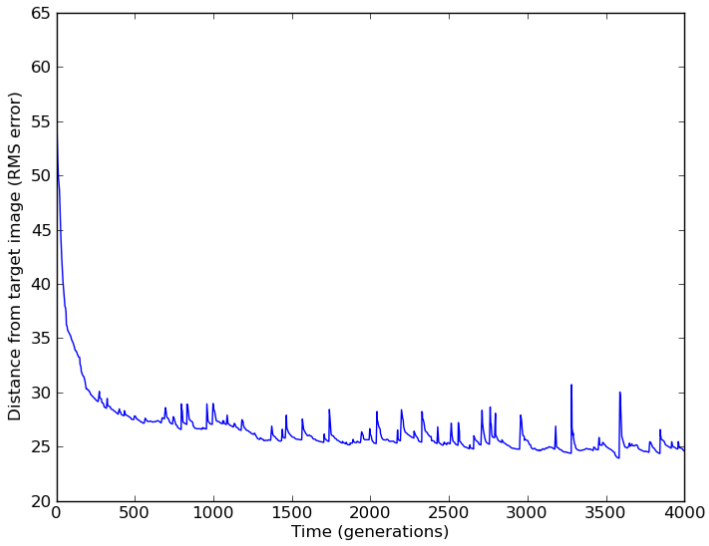


Figure 24. Improvement over time for no subdivision

4.2. Uniform Subdivision

A second attempt was made using uniform subdivision. The image was divided into a uniform 10x10 grid (100 equal-sized sub-images), and this produced significantly better results, as shown in Figure 23. Because of the smaller size and less complex nature of each sub-image, the functions are able to match the sub-images more closely. It should be apparent that we can break the image up into individual pixels in order to obtain the highest quality results, but since we are trying to compress images, this would defeat the purpose of the system.

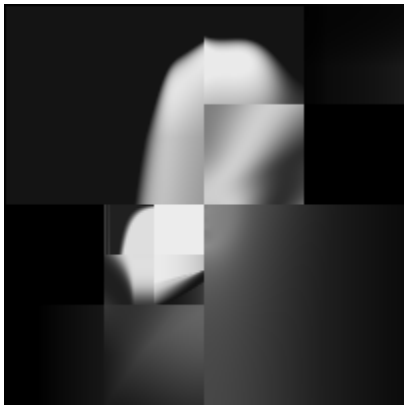
4.3. Variable Subdivision

Recall that variable subdivision allows the system to determine where and how deeply the image should be recursively subdivided based on how well the function matches a given portion of the image. By adjusting for acceptable fitness, we have produced several versions to show that the algorithm is capable of varying accuracy. The sizes for the images are provided as well, based on a simple compression metric:

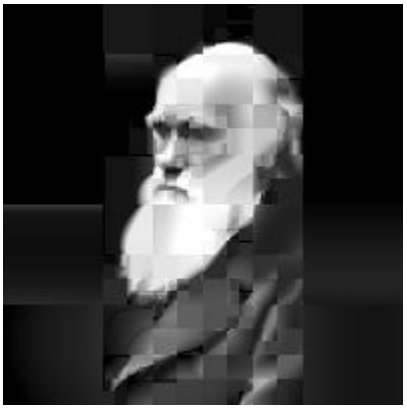
- Operators, names, and parentheses (e.g. “+”, “*x*”, “hypot”) occupy 4 bits
- Floating point numbers occupy 32 bits

Modifying the acceptable fitness for each block of the image allows us to produce variable quality results. It also changes the running time of the algorithm. If the acceptable fitness is chosen to be very high, the algorithm will not descend into smaller sub-images and the quality of the evolved image suffers. However, the resulting set of functions will tend to be smaller and the algorithm finishes much more quickly.

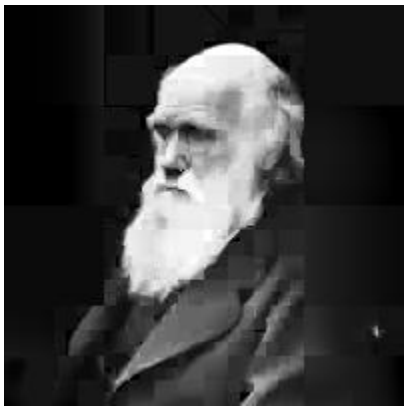
A series of images with varying quality settings is provided below. Because these images are subdivided, we cannot simply show fitness progression graphs for the images, but we show the final RMS error of each image in Figure 25. Due to the subjective nature of image quality, we leave it to the reader to inspect these images and judge how successfully the algorithm currently performs.



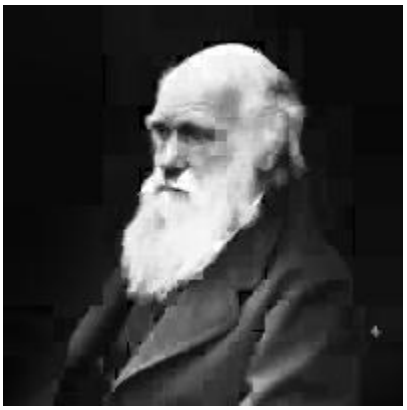
Darwin 1 - Accepted Fitness 32, size 775 bytes



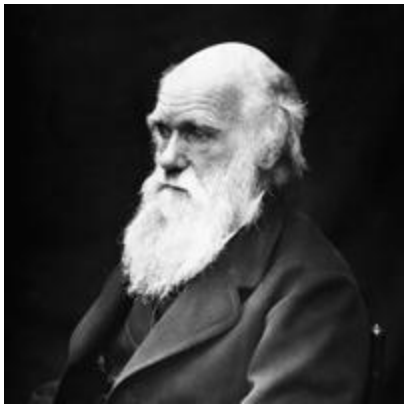
Darwin 2 – Accepted Fitness 16, size 10,234 bytes



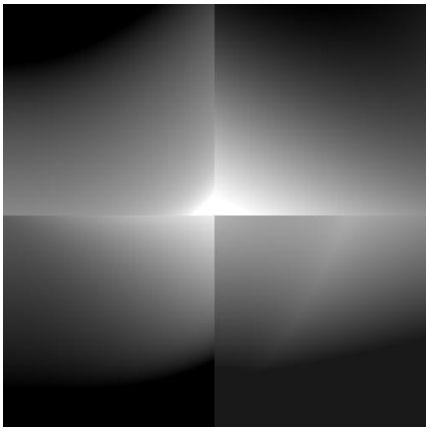
Darwin 3 – Accepted Fitness 8, size 19,571 bytes



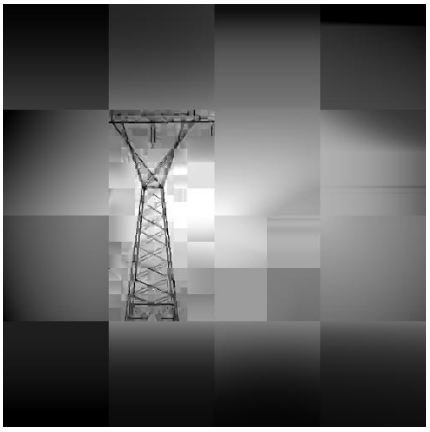
Darwin 4 – Accepted Fitness 4, size 26,242 bytes



Darwin 5 – Uncompressed original image, size 61,084 bytes



Electricity 1 – Accepted Fitness 32, size 170 bytes



Electricity 2 – Accepted Fitness 16, size 17,014 bytes

Electricity 3 – Accepted Fitness 8, size 54544 bytes

Electricity 4 – Accepted Fitness 4, size 103,498 bytes

Electricity 5 – Uncompressed original image, size 393,978 bytes

Our system uses Python 2.6, and has been parallelized to use multiple processors. The runtime (Figure 25) on an Intel Core i7 2.67Ghz 4-core CPU with hyper-threading is from less than 1 hour for the lowest quality "Darwin" image, to 122 hours for the highest quality "Electricity" image.

When the accepted fitness value is high enough, the algorithm will produce mutations that satisfy the requirement without having to recurse. Thus, the resulting error will be less than the accepted fitness. When the accepted fitness is very restrictive, however, the algorithm will often reach its maximum depth and number of generations without satisfying its accepted fitness. This results in the overall RMS error being greater than the accepted fitness value.

Image	Accepted Fitness	Size (bytes)	Running time (hours)	Error (RMS)
Darwin	32	775	< 1	28.17
Darwin	16	10234	9	14.69
Darwin	8	19571	21	9.17
Darwin	4	26242	32	7.54
Electricity	32	170	< 1	29.37
Electricity	16	17014	15	16.67
Electricity	8	54544	73	10.78
Electricity	4	103498	122	9.11

Figure 25. Accepted fitness, size, running time, and RMS error for each image

To compare our algorithm to the current state of the art, we compressed the same images into JPEG using the open-source ImageMagick[2] library with default quality settings. We then calculated the RMS error on the images (Figure 26). This should be compared to

[2] http://www.imagemagick.org/

Figure 25.

Image	Size (bytes)	Error (RMS)
Darwin	8678	1.74
Electricity	50473	1.63

Figure 26. Results from JPEG

Due to the random nature of genetic algorithms, multiple executions will produce different results. The seed for the random number generator could be specifically chosen so that the results are always the same (given the same parameters to the system), but we chose not to do this. Consequently, multiple runs of our system produced different results. These differences can be significant—they are not only visibly different, but they also differ in size from one another (Figures 27 and 28).

Figure 27. Cat; First run, 20,773 bytes

Figure 28. Cat; second run, 21,247 bytes

Note that because we are representing images as expressions, we are able to reconstruct the result at an arbitrarily high resolution. However, it is currently unlikely that a compressed representation of an image will be so accurate that it would warrant an increase in resolution. Perhaps a more advanced version of the algorithm could take advantage of this.

5. Conclusion and Future Work

5.1. Conclusion

In this thesis, we developed a novel application of genetic programming to image compression. We demonstrated that it is possible to evolve program expressions that represent target images with significant reduction in size albeit with some loss of image quality. We showed several initial results, providing a proof of concept that our technique works in principle as a lossy compression method, and proposed that it could be a promising approach to real-world image compression and other multimedia data compression as well.

While evolving the representation of an image takes a lengthy amount of time, rendering the image from the evolved expression is fast enough for real-time processing. However, the processing time can be reduced because the algorithm (both when evolving a representation and when rendering it) is "embarrassingly" parallel and it speeds up linearly with the number of processors added to the system. Should it prove to become a useful means of compression, the cost of running our algorithm may be offset by the benefit of smaller data transfer when an image is to be transferred multiple times or over low-bandwidth connections.

5.2. Future Work

Considerable work remains to be done in order to make genetic programming feasible for real-world image compression, which we discuss next. We anticipate that if this work is done, genetic programming can become a powerful means of multimedia compression.

Our choices of functions and language constructs in representing images are currently limited, and they are probably not ideal; other constructs and functions may prove to produce higher quality compressed images in less time and using less storage. The space of possibilities is large. A different language may also be chosen. While we used Python (excluding many of its language constructs), a special language can potentially be de-

signed just for the purpose of creating image representations. The means of choosing this language and its constructs is as yet unexplored and it deserves special consideration.

Our current choice of fitness function can be improved to give greater weight to subjective quality measurements much like JPEG does, mainly with respect to color and brightness sensitivity in human vision.

The final results can be compressed much better, although little work has been done in this regard due to its dependency on the choice of functions and language constructs. Once these choices are made, a bit-wise representation for the constructs can be designed such that the resulting expressions have minimal size. Mathematical expressions can also be automatically simplified to reduce the size of the final expression. For instance, the expression “5.2 + 3.1” occupies more space than the expression “8.3”, although they represent the same thing. We currently do not perform any simplification on our evolved or evolving expressions.

There is also a need to consider the problem of code growth while the genetic programming algorithm runs. Research has been done on code growth in genetic programming [Dickinson et al., 1996]. Most genetic programming algorithms are not very adversely affected by unnecessary code growth, but in our case the size of the program is of primary concern. Currently, we impose a hard limit on the size and depth of the abstract syntax tree generated by the algorithm, and delete nodes when this limit is reached. The growth of code as the algorithm runs could potentially be minimized using more elaborate methods than we currently use.

For image reconstruction, there are security concerns due to the nature of interpreting expressions of code. Malicious users might produce code for an image that has undesirable side effects. Depending on the allowed programming constructs and the nature of the interpreter or compiler of the chosen language, these side effects may include stack over-

flows, and long or infinite running times. These concerns should eventually be addressed.

In addition to images, audio and video may also be represented in a similar manner. Audio can be represented as a single-variable expression, where the variable is time. Video also requires the addition of a time variable to the existing *x* and *y* image variables. Like our user-generated images in Section 1, we have also implemented user-generated audio and video functions on our website[3]. Unfortunately, the time that it takes to compress an image using our method is already very long, and video is likely to take orders of magnitude longer. It is currently infeasible to compress video without massive parallelism, faster hardware, and/or until work is done to speed up the algorithm. It is not immediately clear how effective genetic programming will be for audio or video compression, but the promise that it shows for image compression indicates that it may perform well for other media formats.

[3] http://fragsworth.com/audio_functions and http://fragsworth.com/video_functions

References

Ahmed, N., Natarajan, T., Rao, K.R. “Discrete cosine transform.” IEEE *Transactions on Computers*, C-23 Issue 1, 1974 pp. 90-93

Alsing, R. “Genetic Programming: Evolution of Mona Lisa.” December 2008. Retrieved January 20, 2010 at http://rogeralsing.com/2008/12/07/genetic-programming-evolution-of-mona-lisa/

Barnsley, M.F. and Demko, S. “Iterated Function Systems and the Global Construction of Fractals.” *Proceedings of the Royal Society of London*, A399, 1985, pp. 243-275.

Barnsley, M.F., Hurd, L.P., and Anson, L.F. *Fractal Image Compression*. A.K. Peters Ltd., 1993.

Dickinson, J., Foster J. A., Soule, T. “Code growth in genetic programming.” *Proceedings of the First Annual Conference on Genetic Programming*. Stanford, 1996. pp. 215-223

Holland, J.H., Goldberg, D.E., “Genetic Algorithms and Machine Learning.” *Machine Learning*, Vol. 3, 1988. pp. 95-99

Koza, J.R., *Genetic programming: On the programming of computers by means of natural selection*, MIT Press, Cambridge, MA, 1992

Rossum, G. *Python Reference Manual*. Python Software Foundation. 2008.

Sims, K. “Artificial Evolution for Computer Graphics.” *ACM SIGGRAPH Computer Graphics*. Vol 25, Issue 4. 1991. pp. 319-328

Wallace, G.K. *The JPEG Still Picture Compression Standard.* IEEE Transactions on Consumer Electronics, Maynard, MA, 1991.

Willis, M.J., Hiden, H.G., Marenbach, P., McKay, B., Montague, G.A. *Genetic Programming: An Introduction and Survey of Applications*. Dept. of Chem. & Process Eng., Newcastle upon Tyne Univ. Glasgow, UK. 1997